In Search of Strong Generalization:
What can we learn from programming languages?

Danny Tarlow
Researcher
Microsoft Research

Joint work with Alex Gaunt, Marc Brockschmidt, Yujia Li, Richard Zemel, Nate Kushman, Chengtao Li.
Motivation - Strong Generalization

Data

Tasks

Small data

Big data

http://www.scriptol.com/robotics/robots/household.php
Motivation - Strong Generalization
Claim: We need diverse data, but also structured models that have surprisingly strong generalization abilities.
1. Strong generalization
   - By building neural network models inspired by natural source code

2. Weak supervision
   - By differentiating through approximate marginalization algorithms

1. Strong generalization
   • By building neural network models inspired by *natural source code*¹

2. Weak supervision
   • By differentiating through approximate marginalization algorithms

---

Write a short natural program to add a and b:

```python
def add(a, b):
    carry = 0
    for i in range(len(a)):
        cur = a[i] + b[i] + carry
        result.append(cur % 10)
        carry = cur // 10
    result.append(carry)
    return result
```
def add(a, b):
    carry = 0
    for i in range(len(a)):
        cur = a[i] + b[i] + carry
        result.append(cur % 10)
        carry = cur / 10
    result.append(carry)
    return result
Claim / Aspiration:

Programming languages are designed to compactly express the computations that people want to perform, and to make it easy for humans to reason about complex computations. Natural source code induces a prior over natural computations, which we can leverage as inductive bias in machine learning models to achieve strong generalization.

Compared to Kolmogorov complexity, Solomonoff induction, AIXI, etc: here we care about the constants and specific details of, e.g., how modern programming languages represent algorithms. Think python, not binary encoding of programs.
Properties of natural programs:

- algorithm structure often (but not always) invariant to data values
- structured loops (for loops vs goto spaghetti), recursion
- locality / sparsity in accessing & modifying data
- modularity / compositionality / abstraction
- organized into reusable libraries, object-oriented programming
- ...
- ...
Write a short natural program to classify an image

```python
def add(a, b):
    carry = 0
    for i in range(len(a)):
        cur = a[i] + b[i] + carry
        result.append(cur % 10)
        carry = cur / 10
    result.append(carry)
    return result
```

Source Code Inductive Bias Not Always Favorable
Goal

Source code inductive bias for strong generalization

+ Neural networks to handle rich data types
Other examples of encoding algorithmic structure into model


How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures
   • Graph algorithms → Graph Neural Networks

b. Learn models represented as source code
   • Extend differentiable interpreters to learn neural network subroutines
How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures
   • Graph algorithms → Graph Neural Networks

b. Learn models represented as source code
   • Extend differentiable interpreters to learn neural network subroutines
Example Graph Algorithm: Bellman Ford

Computes shortest paths from source to all other vertices — works for any graph structure

for v in vertices:
    distance[v] = inf
    predecessor[v] = null
    distance[source] = 0

for i in range(len(vertices)) - 1:
    for (u, v) in edges:
        if distance[u] + 1 < distance[v]:
            distance[v] = distance[u] + 1
            // assume edge costs=1
            predecessor[v] = u

Decode solution from node representations
Graph Neural Networks

- Initialize node representations (hidden vector per node)
- Repeatedly update node representations as learned function of neighbor representations and edge types
- Decode prediction from node representations

End-to-end differentiable, train by SGD
Gated Graph Sequence Neural Networks

Figure 1: (a) Example graph. Color denotes edge types. (b) Unrolled one timestep. (c) Parameter tying and sparsity in recurrent matrix. Letters denote edge types with $B'$ corresponding to the reverse edge of type $B$. $B$ and $B'$ denote distinct parameters.

\begin{align}
\mathbf{h}_v^{\downarrow} &= [\mathbf{x}_v^\top, \mathbf{0}]^\top \\
\mathbf{a}_v^{(t)} &= \mathbf{A}^{\top} \left[ \mathbf{h}_v^{(t-1)} \ldots \mathbf{h}_v^{(0)} \right]^\top + \mathbf{b} \tag{2} \\
\mathbf{z}_v^t &= \sigma \left( \mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)} \right) \tag{3} \\
\mathbf{r}_v^t &= \sigma \left( \mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)} \right) \tag{4} \\
\mathbf{h}_v^{(t)} &= (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \mathbf{h}_v^{(t)} \tag{6}
\end{align}

From Li et al (ICLR 2016).


Simple Reasoning Task

D is A
B is E
A has_fear F
G is F
E has_fear H
H has_fear A
C is H
eval B has_fear H?

Simple Reasoning Task - RNN representation

D is A
B is E
A has_fear F
G is F
E has_fear H
H has_fear A
C is H
eval B has_fear H

n0 is n1
n2 is n3
n1 has_fear n4
n5 is n4
n3 has_fear n6
n6 has_fear n1
n7 is n6
eval n2 has_fear n6

Normalize
Simple Reasoning Task - GNN representation

Properties that enable strong generalization:

Value independence:
Learning propagation algorithm, not mapping of name sequences to answers → generalizes to node names and graph structures never seen during training.

Modular: Resilient to adding “distraction subgraphs”.

Main Limitations:
Not always easy to convert real data (e.g., natural language) to graph format (but see Johnson (2016)) + memory use.

D is A
B is E
A has_fear F
G is F
E has_fear H
H has_fear A
C is H
eval B has_fear H

D is E
is
has_fear

B

is

A

is

D

is

has_fear

has_fear

G

is

H

is

F

is

Gated Graph Sequence Neural Networks - Experiments

Single Output tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>RNN</th>
<th>LSTM</th>
<th>GG-NN</th>
</tr>
</thead>
<tbody>
<tr>
<td>bAbI Task 4</td>
<td>97.3±1.9 (250)</td>
<td>97.4±2.0 (250)</td>
<td>100.0±0.0 (50)</td>
</tr>
<tr>
<td>bAbI Task 15</td>
<td>48.6±1.9 (950)</td>
<td>50.3±1.3 (950)</td>
<td>100.0±0.0 (50)</td>
</tr>
<tr>
<td>bAbI Task 16</td>
<td>33.0±1.9 (950)</td>
<td>37.5±0.9 (950)</td>
<td>100.0±0.0 (50)</td>
</tr>
<tr>
<td>bAbI Task 18</td>
<td>88.9±0.9 (950)</td>
<td>88.9±0.8 (950)</td>
<td>100.0±0.0 (50)</td>
</tr>
</tbody>
</table>

Table 1: Accuracy in percentage of different models for different tasks. Number in parentheses is number of training examples required to reach shown accuracy.

Sequential Output tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>RNN</th>
<th>LSTM</th>
<th>GGS-NNs</th>
</tr>
</thead>
<tbody>
<tr>
<td>bAbI Task 19</td>
<td>24.7±2.7 (950)</td>
<td>28.2±1.3 (950)</td>
<td>92.5±5.9 (100)</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>9.7±1.7 (950)</td>
<td>10.5±1.2 (950)</td>
<td>99.0±1.1 (250)</td>
</tr>
<tr>
<td>Eulerian Circuit</td>
<td>0.3±0.2 (950)</td>
<td>0.1±0.2 (950)</td>
<td>100.0± 0.0 (50)</td>
</tr>
</tbody>
</table>

Table 3: Accuracy in percentage of different models for different tasks. The number in parentheses is number of training examples required to reach that level of accuracy.

**Takeaway** - when problem is well-represented as a simple graph, GNN formulation learns a more accurate model from less data.
How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures
   - Graph algorithms $\rightarrow$ Graph Neural Networks

b. Learn models represented as source code
   - Extend differentiable interpreters to learn neural network subroutines
How to combine source code bias and neural nets?

a. Build models around specific algorithmic structures
   • Graph algorithms → Graph Neural Networks

b. Learn models represented as source code
   • Extend differentiable interpreters to learn neural network subroutines
GNNs are built around a very restricted algorithmic template.

Can we also learn the algorithmic template?

→ Build on differentiable interpreters

Differentiable Interpreter: Example

1. Let **Params** be learnable categorical distributions
2. Lift all operations (squares) to be differentiable
3. Observe discrete inputs and outputs
4. Maximize $p(\text{outputs} | \text{inputs}; \text{Params})$
Lifting Function Applications

\[ \mu_\gamma(\gamma) = \sum_{\alpha, \beta} \mu_\alpha(\alpha) \mu_\beta(\beta) [\gamma = g(\alpha, \beta)] \]

Example:
\[ \begin{align*} 
\alpha &= \text{register 1} \\
\beta &= \text{register 2} \\
\gamma &= \text{result} \\
g &= \text{add} 
\end{align*} \]

Similar logic applies to if statements.

See Gaunt et al. poster on “TerpreT” for lots more.
Adding Neural Function Calls to Differentiable Interpreter

Task: learn to classify dinosaurs from counts supervision

Define some discrete functions

```
Define a neural function = instantiate a neural net

Program to infer: decide whether to MOVE or READ at each timestep t = 1…T

Can then call neural functions like discrete functions, taking tensor data as input

Observe total counts

End-to-end differentiable, train by SGD
```

Lifelong Perceptual Programming by Example
Alex Gaunt, Marc Brockschmidt, Nate Kushman, Daniel Tarlow.
arXiv:1611.02109
Learn programs for sequence of tasks; share neural functions

Resilient to catastrophic forgetting; demonstrates reverse transfer

Distribution of tasks vs time

Last seen example of first task

First task performances (solids) and baseline multitask net (dashed)
Strong Generalization

Figure 7: Generalisation behaviour on MATH expressions of varying length after training on 2 digit expressions.

**Strengths:**
Exhibits strong generalization-error is just based on error of MNIST classification

Learning is cumulative; can continue to update all parameters as task distribution shifts

**Main Limitation:**
Differentiable interpreters are susceptible to local optima. Training requires many random restarts (here ~50)

**Future:**
Incorporate learned priors over source code into model
Why does this mitigate catastrophic forgetting?

Hinton on Mixture of Experts:

“This may allow particular models to specialize in a subset of the training cases. **They do not learn on cases for which they are not picked. So they can ignore stuff they are not good at modeling.**”

Our speculation:

When a network starts to specialize, it provides enough signal to the source code component to know which network to use. Once the source code component picks a network, the others can ignore stuff they are not good at modeling. The source code component *focuses the supervision* for the neural nets.


1. Strong generalization
   • By building neural network models inspired by natural source code$^1$

2. Weak supervision
   • By differentiating through approximate marginalization algorithms

---

1. Strong generalization
   • By building neural network models inspired by *natural source code* \(^1\)

2. Weak supervision
   • By differentiating through approximate marginalization algorithms

---

Weak Supervision

Idea: train Neural Programmer-Interpreter with weaker supervision by building model that shares algorithm structure with dynamic program to compute marginal log likelihood of observations.

Builds on:
- Connectionist Temporal Classification (Graves et al, 2006)
- Stack RNN (Joulin & Mikolov, 2015)

Neural Program Lattices
Chengtao Li, Daniel Tarlow, Alex Gaunt, Marc Brockschmidt, Nate Kushman.
ICLR 2017 Submission
Conclusions

Graph Neural Networks
- It might be natural to encode your data as a graph.
- Node representations can be output of / input to other net to handle perceptual data.
- We can go a long ways by learning models like GNNs that generalize across graph structures.

Source Code Inductive Bias
- Inspires models that can strongly generalize and build up a library of components over time. Not restricted to symbolic data!
- Some surprising benefits like resilience to catastrophic forgetting.
- Need better program induction methods (see Balog et al. “DeepCoder” for using bottom-up cues to aid synthesis)
- Lots more to do in this space!